Pragmatics of Large Values

Real machines are very efficient at handling word-size chunks of data (e.g. 16-64 bits depending on hardware). Things that fit easily in a word:

- Numbers, characters, booleans, enumerations, class tags, etc.
- Memory addresses (locations)

Words are very easy to move, load, store, supply to operations, etc.

But how can we manipulate larger chunks of data, such as records or arrays, which may occupy many words?
Boxing

- Two basic ways to represent large values
  - The **unboxed** representation holds the actual bits of the value, using as many machine words as necessary
  - The **boxed** representation allocates separate storage (the “box”) for the actual bits, and then represents the value by the location of that storage

- Boxes are usually, but not necessarily, stored in the heap
- Boxing may be performed implicitly or explicitly

~textbook: “value” model

~textbook: “reference” model
Boxed vs. Unboxed

Example: an array of 100 (machine) integers

- Unboxed implementation: values occupy 100 consecutive words

- Boxed representation: values occupy 1 word pointer pointing to 100 consecutive words contents

Choice of representation can make a big difference to semantics on operations on the data

- What does assignment mean?

- How does parameter passing work?

- What do equality comparisons mean?
Unboxed Assignment Semantics

- Early languages often used unboxed records and arrays.

  TYPE Employee =
  RECORD
    name : ARRAY (1..80) OF CHAR;
    age : INTEGER;
  END;

  Specifies an unboxed representation, in which value of type Employee will occupy 84 bytes (assuming 1 byte characters, 4 byte integers).

  The semantics of assignment is to copy the entire representation.

  VAR e1, e2 : Employee;
  e1.age := 91;
  e2 := e1;
  e1.age := 19;
  WRITE(e1.age, e2.age);

  Prints 19, 91.
Unboxed representation issues

- This assignment semantics seems simple and appealing, but it has problems:
  - Assignment of a large value is expensive, since lots of words may need to be copied
  - Especially hard to generate efficient code if size of large value is not known statically
Boxed Assignment Semantics

Most modern languages (e.g. Java, Python, Haskell) **box** all values (e.g. objects, records, constructions) that are larger than one word.

These languages naturally use **reference** semantics for assignment: just the pointer is copied, creating an **alias**.

```scala
case class emp(var name: String, var age: Int)
object Bat {
  def main(argv: Array[String]) = {
    val e1 = emp("fred", 91)
    val e2 = e1
    e1.age = 19
    println(e1.age + " " + e2.age)
  }
}
```

prints 19,19

Scala
Explicit Pointers

Languages that use unboxed semantics may also have explicit pointer types to support reference-style operations.

```cpp
struct Emp {
    char name[80];
    int age;
};
Emp *e1 = new Emp();
e1->age = 91;
Emp *e2 = e1;
Emp e3 = *e1;
e1->age = 19;
cout << e1->age << " " << e2->age << " " << e3.age << "\n";
```

prints 19,19,91

In C/C++, struct and class instances are fundamentally unboxed, but programmers usually box them explicitly (using `new` or `malloc`) and manipulate them via pointers.
Varieties of Equality

Languages typically provide some form of built-in equality testing on values. When are two (large) values equal?

Under **structural** equality, values are equal when their contents are equal, bit for bit. (Only sane definition for unboxed values.)

Under **reference** equality, values are equal when their locations are identical.

Reference equality $\Rightarrow$ structural equality, but not vice-versa.

Reference equality may be cheaper to check than structural equality.
Multiple kinds of equality

Some language provide both structural and reference equality, under different names.

They may also provide a standard way for programmer to define equality for a given type in an ad-hoc way.

E.g in Scala:

- the `eq` operator gives reference equality
- the `==` operator invokes a user-defined `equals` method
- for case classes `equals` is pre-defined to be structural equality
Procedures and Functions

- Procedures have long history as essential programming tool

- Low-level view: subroutines let us avoid duplicating frequently-used code

- Higher-level view: procedural abstraction lets us divide programs into components with hidden internals

- Procedural abstractions are parameterized over values and (sometimes) types

- A function is just a procedure that returns a result (or, conversely, a procedure is just a function whose result we don’t care about).
Procedure Activation Data

- Each invocation of procedure is specialized by associated activation data, such as:
  - the actual values corresponding to the formal parameters of the procedure
  - locations allocated for the values of local variables
  - the return address in the caller

- Activation data lives from time procedure is applied until time it returns

- If one procedure calls another, directly or indirectly, their activation data must be kept separate, because lifetimes overlap
  - In particular, each recursive invocation needs new activation data
Activation Stacks

In most languages, activation data can be stored on a stack, and we speak of pushing and popping activation frames on the stack.

```
int f(int x, int y) {
    int z = y+y;
    if (z > 0)
        z = f(z,0);
    return z+y;
}

void main() {
    int w = 10;
    w = f(w,w);
}
```

A typical activation stack, shown just before inner call to `f` returns.
Calling conventions

- In compiled language implementations, we want to be able to generate the code for procedures separately from the code for their applications.
  
  - E.g., procedure may live in a pre-compiled library.

- Requires a calling convention between caller and callee: e.g., caller places parameter values on the stack in a fixed order, and callee looks for them there.

- In an interpreter, where caller and callee are visible at the same time, it is easy to be imprecise about this, but we have been trying to build a careful model.
What about registers?

For simplicity, we view store locations as memory addresses.

But most real machines also have registers, which are:

- much faster to access than memory
- very limited in number (e.g. 4 to 64)
- don’t have addresses, so cannot be accessed via an indirection

Compilers try to keep variables (and pass parameters) in registers when possible, but always need memory as a backup.

Using registers is just an (important!) optimization.
Procedure Parameter Passing

When we apply a function in an imperative language, the formal parameters get bound to locations containing values.

- How is this done and which locations are used?
- Do we pass addresses or contents of variables from the caller?
- How do we pass actual values that aren't variables?
- What does it mean to pass a large value like an array?

Two main approaches: call-by-value (CBV) and call-by-reference (CBR).

Also call-by-name/need (CBN).
Call-by-value

- Each actual parameter is evaluated to a value before call.

- On entry to function, each formal parameter is bound to a freshly-allocated location, and the actual parameter value is copied into that location.

  - Much like processing declaration and initialization of a local variable.

- Semantics are just like assignment of actual expression to formal parameter.

- Simple; easy to understand!
Issues with call-by-value

- Updating a formal parameter doesn’t affect actuals in the caller.
- Usually a good thing!
- But sometimes not what we want

```c
void swap(int i, int j) {
    int t;
    t = i; i = j; j = t;
}
...
swap(a[p], a[q]);
```

Of course, perhaps this is usually a good thing!
Can be inefficient for large unboxed values, e.g. C structs (records):

```c
typedef struct {double a1,a2,...,a10;} vector;
double dotp(vector v, vector w) {
    return v.a1 * w.a1 + v.a2 * w.a2 + ...
        + v.a10 * w.a10;
}
vector v1,v2;
double d = dotp(v1,v2);
```

Call to `dotp` copies 20 doubles
Call-by-reference

- Pass a pointer to the existing location of each actual parameter.

- Within function, references to formal parameter are indirected through this pointer — so parameter can be dereferenced to get the value, but can also be updated.

- If actual argument doesn’t have a location (e.g. is an expression \((x+3)\)) then either:
  - evaluate it into a temporary location and pass address of temporary, or
  - treat as an error.
Issues with Call-by-reference

- Now procedures like \texttt{swap} work fine!

- Can also return values from procedure by assigning to parameters

- Lots of opportunity for aliasing problems, e.g.

```pascal
PROCEDURE matmult(a,b,c: MATRIX)
... (* sets c := a * b *)
matmult(a,b,a) (* oops! *)
```

overwrites parts of argument as it computes result
Hybrid methods

In Pascal, Ada, and C++, programmer can specify (in the procedure header) for each parameter whether to use CBV or CBR

C always uses CBV, but programmers can take the address of a variable explicitly, and pass that to obtain CBR-like behavior:

```c
swap(int *a, int *b) {
    int t;
    t = *a; *a = *b; *b = t; }
swap (&a[p],&a[q]);
```
Values can be References

- In many modern languages, like Java or Python, both records (objects) and arrays are always boxed, so values of these types are already pointers (or references).

- Thus, even if the language uses CBV, the values that are passed are actually references: calls don’t cause any actual copying of the large values.

- But it is a mistake (which some otherwise good authors make) to say that these languages use “call-by-reference” (If they did, they would be passing a reference to the reference!)
Recall that one simple way to give semantics to procedure calls is to say they behave “as if” the procedure body were textually substituted for the call, substituting actual parameters for formal ones.

This is very similar to macro-expansion, which really does this substitution (statically). E.g. (in C):

```c
#define swap(x,y) {int t; t = x; x = y; y = t;}
...
swap(a[p],a[q]);
```

expands to

```c
{int t; t = a[p]; a[p] = a[q]; a[q] = t;}
```
Avoiding capture

• Blind substitution is dangerous!

```c
#define swap(x,y) {int t;t = x;x = y;y = t;}
```

```c
swap(a[t],a[q])
```

expands to

```c
{int t; t = a[t]; a[t] = a[q]; a[q] = t;}
```

Nonsense!

Say $t$ has been captured by the declaration in the macro block.
One solution is to note that names of local variables are not important, e.g. we can rename to

```c
{int u; u = a[t]; a[t] = a[q]; a[q] = u;}
```

Call-by-name can be thought of as “substitution with renaming where necessary”

On real machines, CBN is implemented by passing to the function the AST for actual argument + values of its free variables

This makes CBN much less efficient to implement than CBV or CBR. (We may see more later.)
Call-by-need

A very useful feature of call-by-name is that arguments are evaluated only if needed

\[ \text{foo } x \ y = \begin{cases} x & \text{if } x > 0 \\ y & \text{else} \end{cases} \]

\[ \text{foo } 1 \ (\text{factorial } 1000000) \]

As a further refinement, “pure” functional languages typically use call-by-need (or lazy) evaluation, in which arguments are evaluated at most once:

\[ \text{foo } x \ y = \begin{cases} x & \text{if } x > 0 \\ y \times y & \text{else} \end{cases} \]

\[ \text{foo } (-1) \ (\text{factorial } 1000000) \]